# Discuss query optimization techniques

## Indexes

* An index is a mechanism to boost performance – logical pointer to a physical location
* An index can be placed on a column or columns to help the system find the data faster
* Whenever that data changes, the corresponding index has to be changed as well,
* Every time data in the system is changed, any corresponding indexes have to be updated as well –this is done automatically by the system.

Syntax:

CREATE [ UNIQUE | BITMAP ] INDEX index\_name ON table\_name ( {column\_name | column\_expressions} [ASC | DESC] [, …] ) [various storage attributes];

* BITMAP index – Works best for columns with small set of possible values – i.e. age group (c,t,a,s)
* UNIQUE index – The index prevents duplication of data – the system checks for duplicates when the index is created and each time data is added. PK and unique constraints have these automatically created for them.
* Terms:
  + B-tree index – The default and most common type of index
  + Clustered index – Sorts table rows in the same physical order as the index
  + Composite index – Index involving more than one column – Start with column most used in searches because the search will use all of the indexed columns or just the leading portion in searches/sorts.
  + Function-based index – An index computed column (eg UPPER(lastName)) a
    - An index made up of already calculated values.

# How, what, why to Index

### Advantage of indexing:

* Indexes make queries faster(theoretically, if the query can use them)

### Disadvantages of indexing:

* Building and maintaining an index takes time and storage space on the server
* Inserts, updates, and deletes on indexed columns may take longer because the index has to be updated in addition to the data itself.

### Considerations

* Frequency of queries (reads) versus frequency of updates (writes)
  + More indexes mean faster retrieval but slower writes.
  + More tables through normalization means less redundancy but more overhead when re-joining tables
    - If there are more queries than modifications, consider denormalizing and adding lots of indexes
    - If the opposite is true (lots of modifications), consider lots of tables and few indexes.
* Likelihood that queries and updates are going to be slow
  + How often the indexed column will be used in the WHERE clause or part of a sort/join etc
  + If the column is never used, pointless to make an index on it.
* Whether to use special types of indexes (e.g. bitmap indexes)

### Guidelines

* Use indexes:
  + When querying 10% or less of the rows in the table
  + Foreign key columns that are often used in joins
  + Columns often accessed in sorted order
  + Columns searched for ranges of values might be suitable to a clustered index
* Don’t use indexes:
  + Columns rarely used in queries for sorting/searching/joins
  + Columns that can only have a few possible values – although a BITAMP index might be suitable.
  + Small tables with few rows
  + If you will be retrieving all rows rather than a few rows

# Query Optimization

SQL is non-procedural – you specify what, not how. The DBMS determines how the work will be accomplished.

The DBMS has an optimizer that attempts to choose the most efficient method to execute the statement – for example, which table to access in which order, what indexes to use, etc.

* Rule-based – Optimizer makes decisions based on a set of ranked guidelines – looking at which indexes are available, unique constraints, etc. This is only considers the structure, not the content of the tables.
* Cost-based – Optimizer chooses the cheapest option (in terms of time), based on statistics kept by the system. To be effective, the statistics must be kept up to date.

There are ways to re-phrase some queries to improve the efficiency of the queries – these things are very system specific (e.g. Oracle quite different than SQL server), so check the docs. Examples include:

* Many DBMS will have some syntax for providing suggestions (hints) on how to run the query.
* There are often tools to see how a query will be run.

### General Guidelines:

* If the index is on a column, if the columns is used with a function, the index may not be used.
* A LIKE expression that starts with a wildcard will use an index e.g lastName LIKE ‘%z%’
* Negative prerequisites (!= or NOT combined with other operations) often do not use indexes, since the assumptions is that mose values in the column will not be equal.

# Query Optimization Tips/Rules

* Use a WHERE clause to filter rows
  + If you are only looking for a specific row, do not ask for all of them with (\*), use WHERE.
  + Example: SELECT \* FROM CUSTOMERS
    - Vs
  + SELECT \* FROM CUSTOMERS WHERE cnum IN (2001, 2002);
* Use Table joins instead of multiple queries
  + If you need information form more than one table, you should use a join rather than multiple queries.
  + Example: SELECT cname FROM Customers WHERE cnum = 2004; and then another query SELECT onum FROM Orders WHERE cnum = 2004;
    - Vs
  + SELECT onum, cname FROM CUSTOMERS c JOIN Orders o ON c.cnum = o.cnum WHERE c.cnum = 2004;
  + In general, it is faster to do one complex query then it is to do multiple simple queries.
  + You should join the tables such that the table with the least number of hits is joined last. Ex, tab1 1000 rows, tab2 100 rows, tab3 10 rows, we’d want to join them in that order.
  + Also, avoid joining complex views together in your query. Each view is also a query, so that query will be running and you will have multiple queries instead of one complex query again.
* Use Fully Qualified Column References When Performing Joins
  + Always include table aliases in the prefix of each column reference. This saves the DBMS from searching the table headers to determine which table the field belongs to.
  + Example: SELECT **o.onum, c.cname** FROM CUSTOMERS c JOIN Orders o ON c.cnum = o.cnum WHERE c.cnum = 2004;
* Use indexes on tables
  + Indexes speed up retrieval if designed properly
* Use WHERE rather than HAVING
  + You should use the WHERE clause to filter rows on single row criteria, and the HAVING clause to filter groups on aggregate criteria.
  + Example: SELECT odate, avg(amt) FROM Orders GROUP BY odate HAVING odate IN (’03-OCT-00’, ’04-OCT-00’);
* Use EXISTS rather than IN for correlated subqueries
  + You should use IN to check a value against a list. You should use EXISTS if results came back, or the existence of results in a subquery. Exists typically offers better performance.
* Use EXISTS Rather than DISTINCT
  + You can suppress the display of duplicates using DISTINCT. Whenever possible, do not use DISTINCT if there is another method that gets the same results.

## Comparing Execution Plans

* We can compare the cost of two execution plans together, the one with the lower cost should be more efficient.